デバイスの記事

SystemCによるビヘイビア合成入門

第7回 ビヘイビア合成のDCT設計への適用

桜井 至

今回は，ビヘイビア合成を実際の設計に適用する場合の注意点と，DCT（discreate cosine transform）の設計に適用した例を紹介する．ビヘイビア合成の技術は，ようやく実設計に適用できる段階に達したばかりである．そのため，まずは利用するビヘイビア合成ツールが得意とする設計，例えば演算系（データパス系）が主体の設計に対して適用することを，筆者は推奨している．（編集部）

この連載も今回で7回目になりました．今まで，ビヘイビア合成の機能，設計フロー，ビヘイビア合成可能なSystemC/C++構文，プロトコル定義，テストベンチなど，基本的な構文や機能を解説しました．この連載をお読みのみなさんには，ビヘイビア合成のひととおりの機能や設計フローについて理解していただけたのではないかと思います．

今回は，ビヘイビア合成を実際に適用する際の設計の選択とその注意点，ビヘイビア合成可能な記述の記述例について紹介します．

● ビヘイビア合成導入時の課題は「いつか来た道」

実際の設計にビヘイビア合成を適用できるかどうかは，まだビヘイビア合成を使ったことがない多くの設計者が疑問に感じているところだと思います．例えば，「ビヘイビア合成によって実用に耐えうる品質のRTL（register transfer level）回路が得られるか」，「実際に開発期間を短縮できるか」，「ビヘイビア記述の記述スタイルを容易に習得できるか」，「下流工程で問題が発生した場合に上流で簡単に対応できるか」など，多くの疑問が湧いてくると思います．

同じような疑問は，ゲート・レベル設計から論理合成を用いたRTL設計に移行する際にも数多くありました．それらの疑問は，設計事例が豊富になり，RTL設計に関連したツールや設計環境が充実するに従って解消されていきました．現在では，HDLと論理合成を用いたRTL設計はごくあたりまえの手法になっています．

ところで，RTL設計の初期のころには，まずはどのような設計に論理合成を適用するかということがよく問題になりました．最初は熟練設計者による設計を必要としない，自動化しやすい設計から順次，論理合成を適用しました．そして，実績や使用経験を積んでから，より複雑で難しい仕様の設計に適用していきました．それとともに，論理合成ツールは初期の限定された機能からより高度で複雑な機能を持つようになり，多くの設計へ適用できるようになりました（稿末のコラム「HDLと論理合成の立ち上げ時の思い出」を参照）．

これと同じことは，ビヘイビア合成についても言えそうです．ビヘイビア合成の技術は，ようやく実設計に適用できる段階に達したばかりです．そのため，まず最初はビヘイビア合成が得意とする設計に限定して適用することがたいせつです．もし，不得意であまり効果の期待できない設計に適用した場合，「ビヘイビア合成は使いものにならない」と判断されてしまい，その後の導入が進まなくなることが考えられます．

● アーキテクチャの分割に従って適用ブロックを選択

通常，ビヘイビア合成を実行する前の段階では，システムのアルゴリズムやアーキテクチャの検討，および検証を行います．設計するシステムには，CPUやメモリ，既設計のIPコア（ハードウェアIP），これから開発するハードウェアなどが含まれます．アルゴリズムをソフトウェアとして処理する場合，CPUに対してプログラムを作成します．

一方，ハードウェアとして処理する場合は，既設計のIPコアを再利用するか，新規のハードウェアを設計します．

アーキテクチャ検証やシステム検証に，SystemCのトランザクション・モデルを用いれば，詳細なクロックやバス，制御信号などの仕様やタイミングを考慮せずに，抽象度の高いレベルで検証できます．トランザクション・レベルからのビヘイビア合成は，現在，開発途上にあります．

設計者はアーキテクチャの分割に従って，ビヘイビア合成を適用する機能ブロック（モジュール）を選択します．現状では，RTLよりやや抽象度の高いバス・サイクル精度（BCA：bus cycle accurate）のレベルまでインターフェース仕様を具体化する必要があります．

● 演算系と制御系の二つに分けて考える

ビヘイビア合成の対象とするブロックを，大ざっぱに演算系と制御系の二つに分けて考えてみます．

ここで演算系とは，おもにアルゴリズムが数式に基づいて記述され，入力データに対して算術演算，論理演算，比較演算などの多くの演算を適用して所望の動作を得るような設計を指すものとします．一般にそのような設計はデータパス系とも呼ばれます．演算系のアプリケーションには，画像処理や暗号化，圧縮/伸張処理，符号化/復号化，データ変換などがあります．

一方，制御系とは，おもにクロック動作に従って，データやメモリ，バスなどの入出力制御を行うアプリケーションを指すものとします．このような制御系の設計では，クロックごとの動作を明確に定義する必要があるため，ステート・マシンを中心とした設計になります．制御系のアプリケーションには，メモリ・コントローラ，バス・コントローラ，各種の標準周辺インターフェースなどがあります．

● ビヘイビア合成が得意とする設計はおもに演算系

ビヘイビア合成が得意とする（または適用する利点の多い）設計は，一般には画像処理などの演算系と言えます[注]．数式や演算子を多く含んだアルゴリズム記述に対して，ビヘイビア合成ツールはスケジューリングを実施し，それぞれの演算子に対して，クロック周期やレイテンシ（クロック・サイクル数）などの設計制約を満足するように演算リソースを共有化します．

図1にビヘイビア合成が生成するRTL設計の構造を示します．ビヘイビア合成によって，レイテンシと面積のトレードオフ評価を行うことができます．目標を満足したRTL記述を用いて論理合成を実行し，既存の設計工程に進みます．

演算系の場合，通常，ビヘイビア記述とRTL記述の記述量に大きな差が生じます（ビヘイビア記述のほうが大幅に記述量が少ない）．設計の抽象度をRTLからビヘイビア・レベルに引き上げることで，検証速度が短縮することも期待できます．

現在，ビヘイビア合成の設計事例として発表されているケースの多くは演算系です．

● 制御系に適用する利点は相対的に低い

制御系の設計をSystemCで記述し，それをビヘイビア合成することもできます．制御系のSystemC記述は，クロックに同期した動作を示す記述となります．クロックご

注：中には制御系の合成に強いアルゴリズムを備えるビヘイビア合成ツールもあるようなので，どのようなタイプの回路を得意とするかについては，ツールの開発元に確認していただきたい．

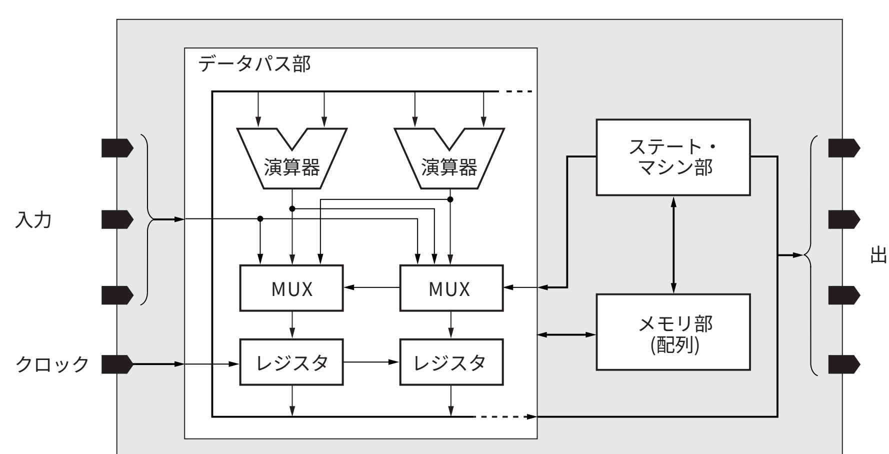
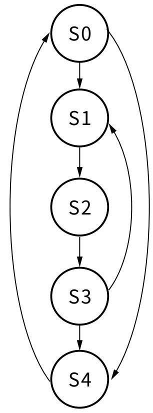


**図1
ビヘイビア合成が生成するRTL設計の構造**

ビヘイビア合成の結果として，演算リソースとレジスタ，およびマルチプレクサ（MUX）から構成されるデータパス部，そのデータパス部を制御するステート・マシン部，配列をメモリにマッピングしたメモリ部で構成されたRTL設計が生成される．

```
void mymod::thread0() {
  {
    CYN_PROTOCOL( “my protocol” );
    while( true ) {
      wait(1);        ←  状態S0
      // 記述0
      for( init; exit_cond; increment ) {
        // 記述1
        wait(1);      ←  状態S1
        // 記述2
        wait(1);      ←  状態S2
        // 記述3
        wait(1);      ←  状態S3
        // 記述4
      }
      // 記述5
    }
      wait(1);        ←  状態S4
      // 記述6
  }
}
```

(a) ステート・マシンの記述



(b) ステート・マシンの構造

図2　ビヘイビア記述による暗黙的なステート・マシンの構造例

制御系のビヘイビア記述はクロックに同期した動作を示す記述となる．クロックごとにどのような動作を実行するかをwait()文に基づいて記述するため，これは暗黙的なステート・マシン記述と言える．このような記述に対して，ビヘイビア合成は明示的な状態を割り付け，ステート・マシンを生成し，さらに必要な演算リソースの割り付けと共有化を行う．

```
always @(posedge(clk))
  ...
    case (global_state)

      `S0 : begin     ←  状態S0
        // 記述0
        // init
        if( !exit cond ) begin
          // 記述1
          global_state <= `S1;
        end
        else
          global_state <= `S4;
       end

      `S1 : begin     ←  状態S1
        // 記述2
        global_state <= `S2
      end

      `S2 : begin     ←  状態S2
        // 記述3
        global_state <= `S3
      end

      `S3 : begin     ←  状態S3
        // 記述4
        // increment
        if( !exit cond ) begin
          // 記述1
          global_state <= `S1;
        end
        else
          // 記述5
          global_state <= `S4;
        end

      `S4 : begin     ←  状態S4
        // 記述6
  ...
```

図3　RTLのVerilog HDL記述による明示的なステート・マシンの構造例

ステート・マシンの構造は図2(b)と同じ，制御系のRTL記述では，明示的に状態名を割り付け，さらに状態の遷移条件を明示的に指定する．記述方法の差異はあるが，ビヘイビア記述と記述量があまり変わらない場合がある．

とにどのような動作を実行するかをwait()文を用いて記述します．このクロックに同期した動作は，暗黙的なステート・マシン記述と言えます．このような暗黙的なステート・マシンに対して，ビヘイビア合成は明示的に状態を割り付け，ステート・マシンを生成し，さらに，必要な演算リソースの割り付けと共有化を行います．

制御系では，ビヘイビア記述とRTL記述の記述量の差があまりありません．例えば，ステート・マシンを暗黙的に記述するか，明示的に記述するかの差しか出ない場合もあります．ビヘイビア記述のステート・マシン記述例の構造と状態遷移を**図2**に，それと等価なRTL記述のステート・マシン記述例の構造を**図3**に示します．記述内に演算が含まれない場合，とくに記述量に大きな差が生じないため，検証速度の向上や記述量の削減による設計期間の短縮などをそれほど見込めません．つまり，制御系にビヘイビア合成を適用する利点は，演算系と比べて少ないと言えます(まったく利点がないわけではないが…)．

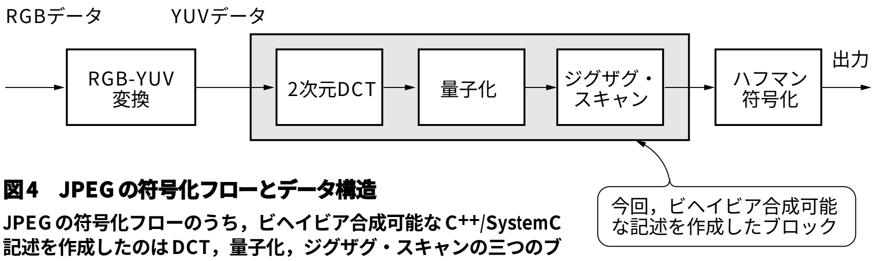
制御系では，さらに，一般にサイクル精度のルールを守る必要があるため，与える制約によって面積を変化させることがほとんどできません．つまり，面積とレイテンシのトレードオフ評価の効果が大きく制限されてしまいます．

以上，ビヘイビア合成を適用するアプリケーションを，大まかに演算系と制御系に分けて考えてみました．演算系だと思っていても，実際には，ほとんどのサイクルがメモリ・アクセスで占められ，制御系に近かったという場合もあります．そのような際には，目標とする性能になかなか到達しないことがあります．また，設計を単純に演算系と制御系に分けられない場合も多いかと思います．

まずは利用するビヘイビア合成ツールが得意とする設計を理解し，そのような設計から適用することが重要です．それによって実績が生まれ，次の段階に進むことができます．現状では，ビヘイビア合成ツールは万能のツールではありません．

● DCTやジグザグ・スキャンなどをSystemCで記述

ビヘイビア合成を適用する演算系の設計例として，DCT(discreate cosine transform；離散コサイン変換)の記述を紹介します．DCTはJPEGなどの画像圧縮アルゴリズム



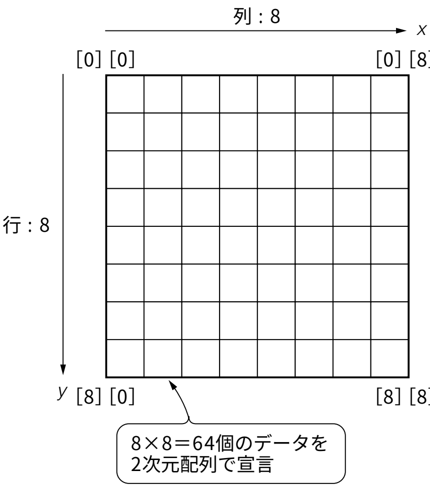


図4　JPEGの符号化フローとデータ構造
JPEGの符号化フローのうち，ビヘイビア合成可能なC++/SystemC記述を作成したのはDCT，量子化，ジグザグ・スキャンの三つのブロック．64個のデータについては，1次元配列ではなく8×8の2次元配列を使用した．

として広く使用されています．DCTには演算式が多いため，ビヘイビア合成に適している設計の一例と言えます．

図4にJPEGの符号化のフローを示します．今回，ビヘイビア合成可能なC++/SystemCで記述したのはDCT，量子化，ジグザグ・スキャンの三つのブロックです．アルゴリズムの詳細については専門書を参照してください．専門書では，配列は1次元配列（例えば`table[64]`）として宣言することが多いようですが，ここでは8×8の2次元配列を使うようにしてみました．

共通ヘッダを**リスト1**に，ビヘイビア合成の対象とするDCTのヘッダを**リスト2**に，アルゴリズム記述を**リスト3**に示します．この記述は，筆者が本連載のために一般的なプログラムや書籍を参考に作成したものです．筆者は画像処理の専門家ではないので，記述内にまちがいがあるかもしれませんが，そのときはご容赦ください．ここで重要なことは，これまでDCTの回路を設計したことのない筆者でも，このようなアルゴリズム的な記述からRTL記述を生成できるという点です．

リスト2のヘッダ・ファイルには，DCT演算で使用するcos（コサイン）値，YおよびUVデータに対する量子化テーブル，スキャン順序を示したテーブルなど，必要な定数を2次元配列として宣言しています．ビヘイビア合成では，定数配列は，ROMにマッピングすることも，組み合わせ回路として実現することもできます．次に`SC_MODULE`を用いて，モジュール名や入出力ポート，メンバ関数，メンバ変数などを宣言しています．`SC_CTOR`コンストラクタには，クロックに同期して起動するプロセスおよびリセットを定義します．ここでは，`thread0`が最上位の関数です．ビヘイビア合成では，プロセスとして`SC_CTHREAD`と`SC_METHOD`を使用できます（`SC_THREAD`は使用できない）．

リスト3のアルゴリズムでは，機能を関数ごとに分割し

リスト1　共通ヘッダ・ファイル（defs.h）

```
// defs.hファイル
#define TRACE_ON 1
#define DUMP_ON  0

#define CLOCK_PERIOD 10

#define Y  1
#define UV 2
```

ています．リストの最後にある関数`thread0`が，最上位の関数となります．関数`thread0`では，SystemC構文を用いてリセットやデータの入力と出力，関数の呼び出しを定義しています．入力データは64個を連続的に入力し，2次元配列変数`block`に保存します．出力ではデータを64個連続的に出力します．入力プロトコルと出力プロトコルの間に演算を実行する関数を呼び出しています．

DCTを実行する関数`fdct()`では，最初の`for`ループで2次元配列の列を計算し，次の`for`ループで行を計算します．DCTの実際の計算は，`calc_dct()`関数によって処理されます．

`quantum`関数では量子化を行います．量子化は，YとUVでは異なる配列を参照します．ここでは，量子化は，単純に割り算を用いて計算しています（実際には，ROMテーブルやそのほかの方法を用いることが多い）．割り算については，データが負の場合は`sc_abs()`を用いて正の値にしてから割り算を行うようにしています．結果は四捨五入されます．

ジグザグ・スキャンを実行する`zigzag`関数では，配列`scan_order`に従ってデータを並べ替えます．2次元配列を用いているため，`index`変数を上位側と下位側のレンジ

リスト2　DCTのヘッダ・ファイル

```
// dct.hファイル
#ifndef _DCT_H_
#define _DCT_H_

#include <systemc.h>
#include “defs.h”

// DCT計算用係数値の定義
#define C1  (sc_uint<16>)(0x16a0)
#define C2  (sc_uint<16>)(0x1d90)
#define C3  (sc_uint<16>)(0x0c3e)
#define C4  (sc_uint<16>)(0x1f62)
#define C5  (sc_uint<16>)(0x1a9b)
#define C6  (sc_uint<16>)(0x11c7)
#define C7  (sc_uint<16>)(0x063e)

// Yデータ用量子化テーブル
const sc_uint<8> y_qt_table[8][8] =
{
  { 8,  6,  5,  8, 12, 20, 27, 31},
  { 6,  6,  7, 10, 13, 29, 30, 28},
  { 7,  7,  8, 12, 20, 29, 35, 28},
  { 7,  9, 11, 15, 26, 44, 40, 31},
  { 9, 11, 19, 28, 34, 55, 52, 39},
  {12, 18, 28, 32, 41, 52, 57, 46},
  {27, 32, 39, 44, 52, 61, 60, 51},
  {36, 46, 48, 49, 56, 50, 52, 50}
};

// UVデータ用量子化テーブル
const sc_uint<8> uv_qt_table[8][8] =
{
  { 9,  9, 12, 24, 50, 50, 50, 50},
  { 9, 11, 13, 33, 50, 50, 50, 50},
  {12, 13, 18, 50, 50, 50, 50, 50},
  {24, 33, 50, 50, 50, 50, 50, 50},
  {50, 50, 50, 50, 50, 50, 50, 50},
  {50, 50, 50, 50, 50, 50, 50, 50},
  {50, 50, 50, 50, 50, 50, 50, 50},
  {50, 50, 50, 50, 50, 50, 50, 50}
};

// ジグザグ・スキャン用順序テーブル
const sc_uint<6> scan_order[8][8] =
{
  { 0,  1,  5,  6, 14, 15, 27, 28},
  { 2,  4,  7, 13, 16, 26, 29, 42},
  { 3,  8, 12, 17, 25, 30, 41, 43},
  { 9, 11, 18, 24, 31, 40, 44, 53},
  {10, 19, 23, 32, 39, 45, 52, 54},
  {20, 22, 33, 38, 46, 51, 55, 60},
  {21, 34, 37, 47, 50, 56, 59, 61},
  {35, 36, 48, 49, 57, 58, 62, 63}
};

// クリップ関数
sc_int<16> clip(sc_int<16> a)
{
  if (a < -2047)
    return (-2047);
  else if (a > 0x7ff)
    return 0x7ff;
  else
    return a;
}


//SystemCモジュールの定義
SC_MODULE(dct) {

 public:
  // 入出力ポート宣言
  sc_in<bool>   clk;
  sc_in<bool>   reset;

  sc_in<sc_int<16> >  dct_in;
  sc_in<sc_uint<1> >  in_valid, out_ready;
  sc_in<sc_uint<2> >  mode;

  sc_out<sc_int<12> > dct_out;
  sc_out<sc_uint<1> > in_ready, out_valid;

  // メンバ変数の宣言
  sc_int<16>  block[8][8];
  sc_uint<2>  yuv_mode;

  // メンバ関数の宣言
  void dump_block();
  void calc_dct(sc_int<16> *pix, sc_int<16> *res);
  void fdct();
  void quantize();
  void zigzag();

  // メイン・スレッド
  void thread0();

  // コンストラクタの宣言
  SC_CTOR(dct) {
    SC_CTHREAD(thread0, clk.pos() );
    watching(reset.delayed() == false);
  }
};

#endif  // _DCT_H_ の終了
```

リスト3　DCTのアルゴリズム・ファイル

```
#include “dct.h”

// 2次元配列データのダンプ表示関数
void dct::dump_block()
{
  for(int y=0; y<8; y++) {
    printf(“Col(%d): “, y);
    for(int x=0; x<8; x++) {
      printf(“%6d “, (int)block[y][x]);
    }
    printf(“¥n”);
  }
}

// 1次元DCT演算の実行関数
void dct::calc_dct(sc_int<16> *p, sc_int<16> *r)
{
  sc_int<17>  sum[4], sub[4];
  sc_int<30>  t[8];

  sum[0] = p[0] + p[7];
  sum[1] = p[1] + p[6];
  sum[2] = p[2] + p[5];
  sum[3] = p[3] + p[4];
  sub[0] = p[0] - p[7];
  sub[1] = p[1] - p[6];
  sub[2] = p[2] - p[5];
  sub[3] = p[3] - p[4];

  t[0] = C1*((sum[0] + sum[3]) + (sum[1] + sum[2]));
  t[4] = C1*((sum[0] + sum[3]) - (sum[1] + sum[2]));
  t[2] = C2*(sum[0] - sum[3]) + C3*(sum[1] - sum[2]);
  t[6] = C3*(sum[0] - sum[3]) - C2*(sum[1] - sum[2]);

  t[1] = C4*sub[0] + C5*sub[1] + C6*sub[2] + C7*sub[3];
  t[3] = C5*sub[0] - C7*sub[1] - C4*sub[2] - C6*sub[3];
  t[5] = C6*sub[0] - C4*sub[1] + C7*sub[2] + C5*sub[3];
  t[7] = C7*sub[0] - C6*sub[1] + C5*sub[2] - C4*sub[3];

  for (sc_uint<4> i=0; i<8; i++) {
     r[i] = (sc_int<16>)(t[i] >> 14);
  }

}

// 2次元DCT演算の実行関数
void dct::fdct()
{
```

```
  sc_int<16> pix[8];
  sc_int<16> res[8];

  sc_uint<4> x, y;

  // 行データのDCT演算
  for(y=0; y<8; y++) { // 行
    for(x=0; x<8; x++) { // 列
       pix[x] = block[y][x];
    }

    // DCT演算の実行関数の呼び出し
    calc_dct(pix, res);

    for(x=0; x<8; x++) {
       block[y][x] = res[x];
    }
  }

#if DUMP_ON
  printf("¥n** Calculation Result of DCT ROW **¥n");
  dump_block();
#endif

  // 列データのDCT演算
  for(x=0; x<8; x++) { // 列
    for(y=0; y<8; y++) { // 行
       pix[y] = block[y][x];
    }

    // DCT演算の実行関数の呼び出し
    calc_dct(pix, res);

    for(y=0; y<8; y++) {
       block[y][x] = res[y];
    }
  }

#if DUMP_ON
  printf("¥n** Calculation Result of DCT COL **¥n");
  dump_block();
#endif

}

// 量子化の実行関数
void dct::quantize()
{
  sc_int<16> res;
  sc_uint<16> uval;
  sc_uint<8> qt, dither;
  sc_uint<4> x, y;

  for(y=0; y<8; y++) { // 行
    for(x=0; x<8; x++) { // 列
      // 量子化テーブルの呼び出し
      if(yuv_mode == Y) {
        qt = y_qt_table[y][x];
      } else {
        qt = uv_qt_table[y][x];
      }

      // 四捨五入のための計算
      dither = qt >> 1;
      uval = sc_abs(block[y][x]) + dither;
      // 量子化の実行(割り算)
      uval /= qt;
      res = (block[y][x] > 0) ? (sc_int<16>)uval
                : (sc_int<16>)((-1)*uval);
      // 計算値のクリップ
      block[y][x] = clip(res);
    }
  }

#if DUMP_ON
  printf("¥n** Calculation Result of Quantize **¥n");
  dump_block();
#endif
}

// ジグザグ・スキャンの実行関数
void dct::zigzag()
{
  sc_int<16> tmp[8][8];
  sc_uint<6> index;
  sc_uint<4> x, y;

  for(y=0; y<8; y++) { // 行
    for(x=0; x<8; x++) { // 列
      // スキャン用テーブルの呼び出し
      index = scan_order[y][x];
      // 並び換えの実行
      tmp[index.range(5,3)][index.range(2,0)]
           = block[y][x];
    }
  }

  for(y=0; y<8; y++) {  // 行
    for(x=0; x<8; x++) { // 列
         block[y][x] = tmp[y][x];
    }
  }

#if DUMP_ON
  printf("¥n** Calculation Result of Zigzag **¥n");
  dump_block();
#endif
}

// メイン・スレッド(最上位関数)
void dct::thread0()
{
  sc_uint<4> x, y;

  // リセットの実行:
  //  CYN_PROTOCOL()は Forte社Cynthesizer で
  //  サイクル精度を保つことを指定するディレクティブ
  { CYN_PROTOCOL("reset");
    in_ready = 0;
    out_valid = 0;
    wait();
  }

  while(true) {

    { CYN_PROTOCOL("read");
      in_ready = 1;
      do { wait(); } while(!in_valid.read());
      yuv_mode = mode.read();
      in_ready = 0;

      // データの読み込み(64データ)
      for(y=0; y<8; y++) { // 行
        for(x=0; x<8; x++) { // 列
          block[y][x] = dct_in.read();
             wait();
        }
      }
    }

    // 関数の呼び出し
    fdct();        // DCT演算の実行
    quantize();    // 量子化の実行
    zigzag();      // ジグザグ・スキャンの実行

          // データの書き出し(64データ)
    { CYN_PROTOCOL("write");
      do { wait(); } while(!out_ready.read());
      out_valid = 1;
      for(y=0; y<8; y++) {  // 行
        for(x=0; x<8; x++) {  // 列
          dct_out = (sc_int<12>)block[y][x];
             wait();
        }
      }
      out_valid = 0;
    }

  } // whileループの終了
}
```

に分けて配列に指定することで，並び替えを実行しています．

それぞれの関数の #if ～ #endif 文内には，途中の計算結果を表示するダンプ関数が含まれています．**リスト1** にある共通ヘッダ・ファイルの DUMP_ON マクロを

```
#define DUMP_ON 1
```

とすることで，ダンプ関数がコンパイルされ，表示が有効になります．

参考のために，検証に使用したテストベンチ記述を

リスト4　テストベンチ記述例

```
// tb.hヘッダ・ファイル
#ifndef _DCT_TB_H_
#define _DCT_TB_H_

#include <systemc.h>
#include "defs.h"

SC_MODULE(testbench) {

  // 入出力ポートの宣言
  sc_in<bool>  clk;
  sc_out<bool> reset;

  sc_in<sc_int<12> >  dct_out;
  sc_in<sc_uint<1> >  in_ready, out_valid;

  sc_out<sc_int<16> > dct_in;
  sc_out<sc_uint<1> > in_valid, out_ready;
  sc_out<sc_uint<2> > mode;

  // ファイル変数の宣言
  ifstream fi;
  ofstream fo;

  // メンバ変数の宣言
  double start_time, end_time;

  // メンバ関数の宣言
  void source();
  void sink();

  // コンストラクタの宣言
  SC_CTOR(testbench) {
    SC_CTHREAD(source, clk.pos());
    SC_CTHREAD(sink, clk.pos());

    // ファイルのオープン
    fi.open("input.dat");
    fo.open("output.dat");

    // ファイル・オープンのチェック
    if (fi.fail()) {
      cerr << "Can't open file : input.dat" << endl;
      sc_stop();
    }
    if (fo.fail()) {
      cerr << "Can't create file : output.dat" << endl;
      sc_stop();
    }
  }

  // デコンストラクタの宣言
  ~testbench() {
      fi.close();
      fo.close();
  }
};

#endif // End of _DCT_TB_H_


//  tb.cppファイル
#include "tb.h"
#define N_BLOCK 1

void testbench::source()
{
  int data;
  reset = 0;
  in_valid = 0;
  wait(2);

  reset = 1;
  wait(1);

  // 動作開始時間の取得
  start_time = sc_simulation_time();

  for(int k=0; k<N_BLOCK; k++) {

    for(int j=0; j<4; j++) {
      do {wait(); } while(!in_ready.read());
      if(j==0 || j==1)
         mode = Y;
      else
         mode = UV;

      in_valid = 1;

      for(int i=0; i<64; i++) {
       // ファイルからのデータの読み込み
        fi >> data;
      if(fi.fail()) {
          cerr << "Can't read data" << endl;
        sc_stop();
        }
        dct_in = data;
        wait();
      }

    in_valid = 0;
    }
  }
}

void testbench::sink()
{

  for(int k=0; k<N_BLOCK; k++) {

    for(int j=0; j<4; j++) {
      out_ready = 1;
      do {wait(); } while(!out_valid.read());
      out_ready = 0;

      for(int i=0; i<64; i++) {
        // ファイルへのデータの書き出し
        if(out_valid.read())
          fo << i << "  " << dct_out.read() << endl;
        wait();
      }
      fo << endl;
    }
  }

  // 動作終了時間の取得
  end_time = sc_simulation_time();
  // レイテンシの計算
  int latency = (int)(end_time - start_time);
  cout << "Total latency = ";
  cout << latency/CLOCK_PERIOD/N_BLOCK/4 << endl;

  // シミュレーションの終了
  sc_stop();
}
```

リスト4に，階層接続とクロックを定義した記述を**リスト5**に示します．テストベンチは，単にファイルからデータを読み込んで，結果をファイルに落とします．sc_main()がSystemCの最上位となり，sc_main()からトップ階層をインスタンス化しています．sc_main()から複数のインスタンスを呼び出すと，階層構造を変更したり構築したりする際に扱いにくくなるので，トップ階層を作成することをお勧めします．sc_main()は，トップ階層の呼び出しと，シミュレーションを制御するSystemC構文を定義しています．

● パイプライン化してデータを連続出力

0.13μmルールのASICライブラリを用いて，クロック周波数100MHz（10ns）と400MHz（2.5ns）の制約に対してビヘイビア合成を行ってみました．結果のレイテンシと面積の関係を**図5**に示します（面積は正規化している）．レイ

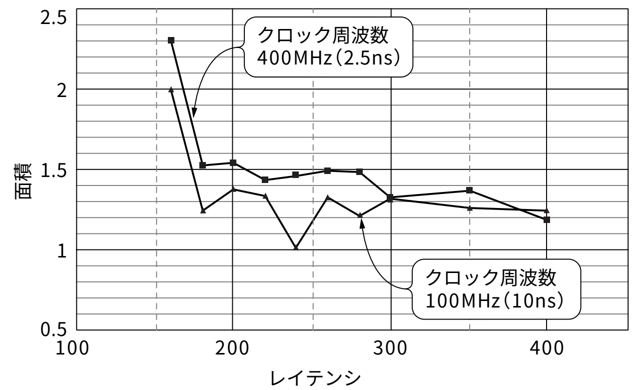


図5　ビヘイビア合成によるレイテンシと面積のトレードオフ評価

ビヘイビア合成ツールを用いてビヘイビア合成した結果のレイテンシと面積の関係を示す．クロック周波数は100MHzと400MHzの2種類．面積は最小面積の結果で正規化している．また，レイテンシについては，データ入力の開始からすべてのデータが出力し終わるまでのサイクル数を示す．ビヘイビア合成では，短時間にレイテンシと面積の異なる複数のRTL構造を生成でき，図のそれぞれが異なるRTLアーキテクチャに対応する．なお，ビヘイビア合成ツールには米国Forte Design Systems社製のものを利用した．

リスト5　階層接続およびsc_main()

```
#include <systemc.h>
#include “defs.h”
#include “tb.h”
#include “dct.h”

SC_MODULE(SYSTEM)
{
 public:
  // クロックの宣言
  sc_clock  clk;

  // 信号の宣言
  sc_signal<bool> reset;
  sc_signal<sc_uint<1> > in_ready, in_valid;
  sc_signal<sc_uint<1> > out_ready, out_valid;
  sc_signal<sc_uint<2> > mode;
  sc_signal<sc_int<16> > dct_in;
  sc_signal<sc_int<12> > dct_out;

  // インスタンス・ポインタの宣言
  dct  *i0;
  testbench  *t0;

  // コンストラクタ宣言
  SC_CTOR(SYSTEM):
     clk(“clock”, CLOCK_PERIOD, 0.5, 0, false)
  {
    // dctインスタンスの呼び出しと接続
    i0 = new dct(“DCT”);
    i0->clk(clk);
    i0->reset(reset);
    i0->in_ready(in_ready);
    i0->in_valid(in_valid);
    i0->out_ready(out_ready);
    i0->out_valid(out_valid);
    i0->mode(mode);
    i0->dct_in(dct_in);
    i0->dct_out(dct_out);

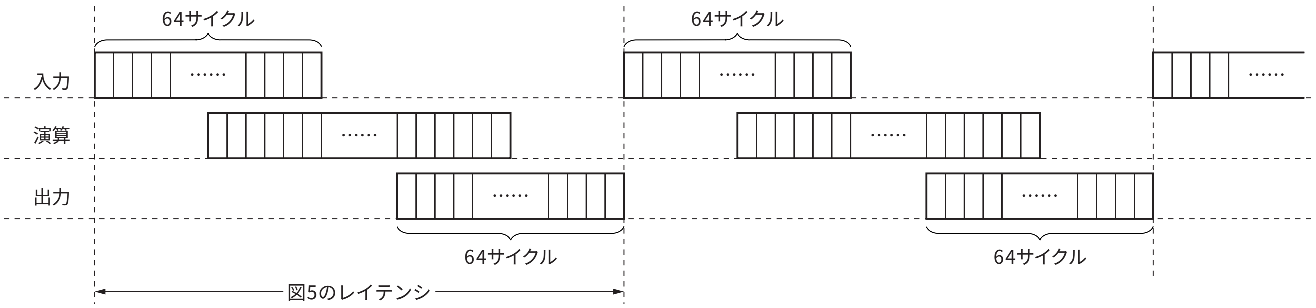
    // testbenchインスタンスの呼び出しと接続
    t0 = new testbench(“TB”);
    t0->clk(clk);
    t0->reset(reset);
    t0->in_ready(in_ready);
    t0->in_valid(in_valid);
    t0->out_ready(out_ready);
    t0->out_valid(out_valid);
    t0->mode(mode);
    t0->dct_in(dct_in);
    t0->dct_out(dct_out);

#if defined TRACE_ON
    // vcdダンプ・ファイルの宣言と信号の指定
    sc_trace_file *vf =
        sc_create_vcd_trace_file(“dump”);
    sc_trace(vf, clk,        “clk”);
    sc_trace(vf, reset,      “reset”);
    sc_trace(vf, in_ready,   “in_ready”);
    sc_trace(vf, in_valid,   “in_valid”);
    sc_trace(vf, out_ready,  “out_ready”);
    sc_trace(vf, out_valid,  “out_valid”);
    sc_trace(vf, mode,       “mode”);
    sc_trace(vf, dct_in,     “dct_in”);
    sc_trace(vf, dct_out,    “dct_out”);
#endif
  }

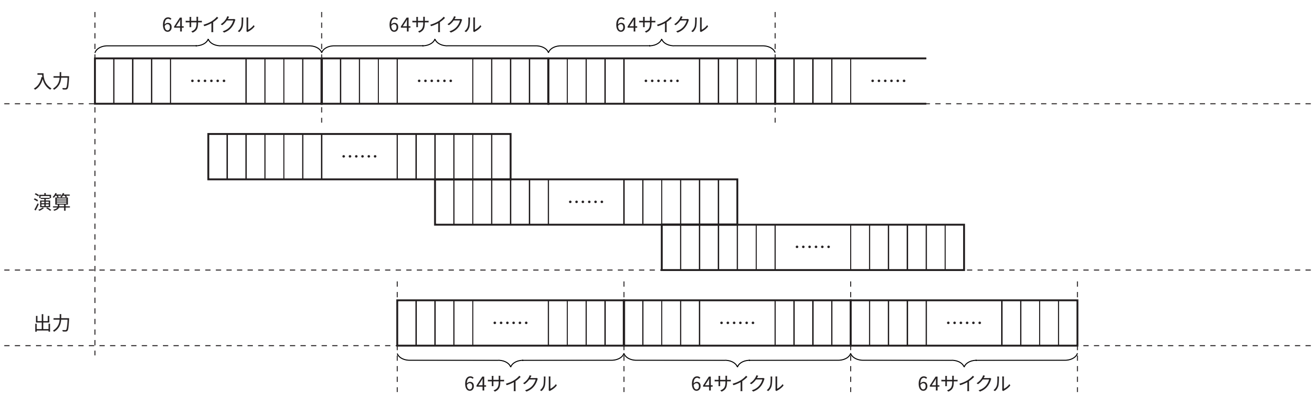
  // デコンストラクタ
  ~SYSTEM() {
     delete i0;
     delete t0;
  }
};


SYSTEM *top = NULL;

// sc_mainの呼び出し（SystemCの最上位）
int sc_main(int argc, char *argv[])
{
    // シミュレーション時間ユニットの指定
    sc_set_default_time_unit(1, SC_NS);
   // 最上位モジュールの呼び出し
    top = new SYSTEM(“top”);
    // シミュレーションの開始
    sc_start();
    return 0;
};
```



(a) 元のサイクル



(b) パイプライン化した場合のサイクル

図6 DCTの元のサイクルとパイプライン化したサイクル
オリジナルの記述をビヘイビア合成すると，(a)のような入出力サイクル・タイミングが得られる．パイプライン化を指定することで，(b)のように連続的にデータを入出力するサイクル・タイミングが得られる．

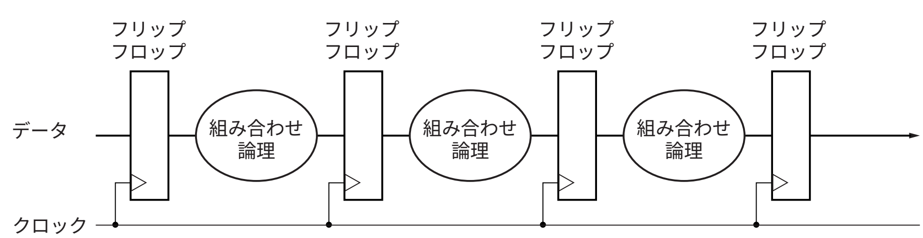


図7
パイプラインの構造
パイプライン化を指定すると，データがレジスタ間を流れながら処理されるような構造が得られる．これはRTL設計ではごく普通の構造だが，ビヘイビア合成では特別な処理が必要になる．図1の構造と比較すると違いが明白．

テンシは，データを入力し始めてからデータがすべて出力し終わるまでとしました．このトレードオフ評価の結果から適切なRTL設計を選択し，論理合成を経てレイアウト設計へと進みます．ビヘイビア合成では短時間に複数のRTL記述を生成させることができます(それぞれのポイントについて，数分程度かかる)．

通常，SystemC記述をビヘイビア合成すると，その結果は，データ入力，演算，データ出力の繰り返しになります．しかし，画像処理の場合，データを連続的に入力し，連続的に出力したいことがあります．その場合には，パイプライン化を指定することで実現できます(**図6**)．ここではデータ入力が64個なので，64サイクル終了後に連続して次の処理を開始するように指定します．それによって，連続的にデータが読み込まれるようになります．

パイプライン化と言うと，なにか特別な処理のように感じられるかもしれません．しかし，RTL設計ではデータをレジスタ間に転送しながら演算処理を定義することが多い(**図7**)ので，パイプラインということばを意識していなくても，実はそのような設計を行っていることが多いのです．

記述例をパイプライン化すると演算に必要なリソースが増えるので，**図5**の結果より面積は約2倍から3倍に増えますが，連続的な処理を行えることが確認できました．

COLUMN　HDLと論理合成の立ち上げ時の思い出

最近，顧客を訪問してSystemCとビヘイビア合成の紹介を行っていると，20年ほど前，筆者がEDA業界に入って，HDLと論理合成を担当し始めたころの状況とよく似ているなと感じます．

筆者が初めて米国でVerilog HDLのトレーニングを受講したのは1987年でした．当時は，論理合成ツールはまだなくて，HDLは設計検証のための言語でした．用途としては，おもにCPU（日本ではとくにTRONチップ）の動作をHDLで記述し，動作検証を行った後，ゲート・レベルで設計を行っていました．これは，SystemCがまずは検証用の言語として使用され，実装の段階ではRTLのHDLで設計している現在の状況と似ています．

また，米国Synopsys社で論理合成ツールのトレーニングを受講したのは1988～1989年ごろでした．当時のSynopsys社の社員は全世界に約50人で，その後，Synopsys社がこれほど大きな会社になるとは，その当時は想像もできませんでした．

今では信じられないかもしれませんが，初期の論理合成ツールはHDLを読み込むことができませんでした．論理ネットリストを読み込んで，論理最適化やテクノロジ・マッピングを行う程度でした．それでも，過去のネットリストを新しい半導体プロセス向けに変換したり，人手で設計した回路に論理最適化を施して，タイミングや面積を改善する用途などに使用されるようになりました．

しばらくして，HDLが読み込めるようになりましたが，最初はレジスタ推定ができなかったため，レジスタはインスタンス化していました．また，算術演算子に対応する演算リソースもサポートしていなかったため，論理合成は，論理演算子と条件文で記述可能な設計にしか適用できませんでした．そのため，RTL設計は，まずは論理式で記述可能な単純な設計から始まりました．その後，論理合成ツールの機能改善が進み，レジスタ推定や演算リソースのサポートも可能となり，現在の記述スタイルが確立されました．

論理合成導入の初期のころ，設計者から出てきた疑問や不安は，今と同じようなものでした．ツールが自動的に生成した設計がほんとうに実用的に使用できるのか，ほんとうに設計期間の短縮につながるのか，などなど…．しかし，論理合成ツールが徐々に普及していくにしたがって，そのような疑問や不安の声は消えていきました．その後，10数年の年月で，グラフィックス入力ツール，記述のリント・チェッカ，フォーマル・ベリフィケーション，フィジカル合成，ECO（engineering change order）などの多くの環境が整い，それによりRTLの標準設計フローが確立し，現在では，HDLと論理合成を用いたRTL設計があたりまえの作業工程になりました．

これと同じことがビヘイビア合成にも言えると思います．ビヘイビア合成技術は，まだまだ未熟な部分があります．しかし，長い目で見て，まずは使用効果の期待できる設計から適用し，実績や経験を積んでいく必要があります．将来的にRTL設計がビヘイビア設計に移行していくことに疑問を感じている設計者は少ないようです．現状は，いつからビヘイビア設計に移行するかを，周囲の状況を見ながら考えているようです．ビヘイビア合成が普及するにしたがって周辺環境が整備され，その後，本格的に普及すると筆者は信じています．

ところで，筆者がEDA業界に入ったころに生まれた方が新入社員として設計のしごとに就くような歳になってきました．月日がたつのは早いものです．設計技術の分野もつねに新しい技術が導入され，動きの速い世界です．慎重さは必要なのですが，保守的になりすぎるのも考えものです．ときには，新しい設計技術に積極的にチャレンジしてみてはいかがでしょうか．

＊　　＊　　＊

今回は，ビヘイビア合成の適用について考えてみました．ビヘイビア合成を適用する際には，まずは，そのツールが得意とするところから適用し，順次，適用する範囲を広げていくことが重要です．

参考・引用*文献

(1) OSCI（Open SystemC Initiative）；*SystemC User's Guide / Functional Specification*，2004.
(2) 長谷川裕恭，小川丈博；連載・SystemCを使ったシステム設計，Design Wave Magazine，2003年7月号～2004年7月号.
(3) Thorsten Grotker，Stan Liao，Grant Martin，Stuart Swan（柿本勝，河原林政道，長谷川隆 監訳）；SystemCによるシステム設計，丸善，2003年.
(4) 大村正之，深山正幸；C/C++によるVLSI設計，共立出版，2003年.
(5) 小野定康，鈴木純司；わかりやすいJPEG/MPEG2の技術，オーム社，2001年.

さくらい・たかし
フォルテ・デザイン・システムズ（株）

<筆者プロフィール>
桜井 至．EDA業界に入り，Verilog HDLから始まり，VHDL，Verilog-AMSへと進み，現在はSystemCを担当．さらに合成ツールとしては，論理合成，アナログ合成と進み，現在はSystemCとビヘイビア合成に従事．HDLと合成の大好き人間と言えるかもしれません．著書に「HDL設計入門」，「HDLによるディジタル設計の基礎」，「LSI設計の基礎技術」（テクノプレス社），および「Verilog-AMS入門」（CQ出版社）がある．現在，フォルテ・デザイン・システムズ（http://www.forteds.com/）に所属．